

# Splitting dataset into training and testing sets

## Why need splitting?

Splitting dataset into training and test sets is a crucial step in machine learning. It allows:

1. Training the model on one part of the data.
2. Testing it on unseen data to evaluate its generalization ability.

Without this split, we risk overfitting, where the model learns patterns specific to the training set but fails to generalize to new data.

## Before splitting dataset

Before we perform any operation on the dataset, we need to encode and clean the dataset first.

- Data encoding is necessary because machine learning algorithms typically work with numerical data, and many datasets contain categorical variables (text-based data). Encoding transforms these categorical variables into numerical representations that the algorithms can understand and process.
- Data cleaning involves identifying and correcting or removing errors, inconsistencies, and inaccuracies in the dataset. It is a crucial step to ensure that the data used for training the machine learning model is of high quality.

## Encode categorical variables:

```
# -----Step 2: Encode categorical variables
# Perform one-hot encoding on categorical columns
data_en = pd.get_dummies(data, columns=['gender', 'Dependents', 'PhoneService', 'MultipleLines', 'InternetService', 'Contract', 'Churn'], drop_first=True)
data_encoded = data_en.map(lambda x: 1 if x is True else (0 if x is False else x))
print(data_encoded)
# -----
```

## Result:

```
"D:\python\Customer churn rate analysis\venv\Scripts\python.exe" "D:\python\Customer churn rate analysis\analysis.py"
SeniorCitizen  tenure  MonthlyCharges  gender_Male  Dependents_Yes  PhoneService_Yes  MultipleLines_Yes  InternetService_Fiber optic  Contract_One year  Contract_Two year  Churn_Yes
0             0       1             29.85           0             0             0             0             0             0             0
1             0      34             56.95           1             0             1             0             0             1             0
2             0       2             53.85           1             0             1             0             0             0             1
3             0      45             42.30           1             0             0             0             0             1             0
4             0       2             70.70           0             0             1             0             1             0             1
...         ...         ...         ...         ...         ...         ...         ...         ...         ...         ...
7038          0      24             84.80           1             1             1             1             0             1             0
7039          0      72             103.20          0             1             1             1             1             1             0
7040          0     11             29.40           0             1             0             0             0             0             0
7041          1       4             74.40           1             0             1             1             1             0             1
7042          0     66             105.65          1             0             1             0             1             0             1
[7043 rows x 11 columns]
```

## Data cleaning:

### Check for missing data:

```
# -----Step 3: Check for missing data
missing_data = data.isnull().sum()
# Display number of missing values in each column
print(missing_data)
# -----
```

**Result:** No missing data is found

```
gender          0
SeniorCitizen   0
Dependents       0
tenure          0
PhoneService    0
MultipleLines    0
InternetService  0
Contract        0
MonthlyCharges  0
Churn           0
dtype: int64
```

**Check for duplicates:**

```
# Check for duplicate rows in the dataset
duplicate_count = data_encoded.duplicated().sum()
print(f"Number of duplicate rows before cleaning: {duplicate_count}")
# If duplicates are found, remove them
✓ if duplicate_count > 0:
    # Remove duplicate rows
    data_encoded_cleaned = data_encoded.drop_duplicates()
    print(f"Number of rows after removing duplicates: {data_encoded_cleaned.shape[0]}")
✓ else:
    # If no duplicates are found, keep the original DataFrame
    data_encoded_cleaned = data_encoded.copy()
    print("No duplicate rows found.")
# Display the cleaned DataFrame's shape
print(f"Shape of the cleaned dataset: {data_encoded_cleaned.shape}")
print(data_encoded_cleaned)
```

**Result:** there are 103 duplicate rows identified. After dropping all the duplicate rows, we have a new dataset that has 6490 rows (originally it has 7043 rows)

```

Number of duplicate rows before cleaning: 103
Number of rows after removing duplicates: 6940
Shape of the cleaned dataset: (6940, 11)
SeniorCitizen  tenure  MonthlyCharges  gender_Male  Dependents_Yes  PhoneService_Yes  MultipleLines_Yes  InternetService_Fiber optic  Contract_One year  Contract_Two year  Churn_Yes
0             0       1         29.85           0             0             0             0             0             0             0
1             0      34         56.95           1             0             1             0             0             1             0
2             0       2         53.85           1             0             1             0             0             0             1
3             0      45         42.30           1             0             0             0             1             0             0
4             0       2         70.70           0             0             1             0             1             0             1
...
7038          0      24         84.80           1             1             1             1             0             1             0
7039          0      72        103.20           0             1             1             1             1             1             0
7040          0      11         29.60           0             1             0             0             0             0             0
7041          1       4         74.40           1             0             1             1             1             0             1
7042          0      66        105.65           1             0             1             0             1             0             1
[6940 rows x 11 columns]

```

## How to split dataset?

We use **train\_test\_split()** function in **scikit-learn** to split the dataset. The **train\_test\_split()** function is part of the **sklearn.model\_selection** module. It splits the data into random subsets (training and testing). Below's a detailed breakdown of its parameters:

### Syntax:

```

sklearn.model_selection.train_test_split(*arrays, test_size=None, train_size=None,
random_state=None, shuffle=True, stratify=None)

```

[\[source\]](#)

(Screenshot taken from [https://scikit-learn.org/stable/modules/generated/sklearn.model\\_selection.train\\_test\\_split.html](https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html))

### Implementation:

```

from sklearn.model_selection import train_test_split
X = data_encoded_cleaned.drop(labels='Churn_Yes', axis=1) # Features (everything except 'Churn_Yes')
y = data_encoded_cleaned['Churn_Yes'] # Target (the column we are trying to predict)
X_train, X_test, y_train, y_test = train_test_split(*arrays: X, y, test_size=0.2, random_state=42)

```

### Parameters

- **X**: This is the feature set (all input columns excluding the target).
- **y**: This is the target variable (the column we are trying to predict, e.g., Churn\_Yes).
- **test\_size**: The proportion of the dataset to include in the test split. Common values:
  - **0.2** means **20% of the dataset will be used for testing** and **80% for training**.
  - It can be adjusted to 0.3 (30% for testing) or any other value depending on your dataset size and problem complexity.
  - A smaller test size (e.g., 0.1) is good if we have a small dataset and want to maximize training data.
    - If we have a small dataset, it's important not to allocate too much data to the test set, as we might not have enough data to train the model properly.
  - A larger test size (e.g., 0.3 or higher) is appropriate if we have enough data and want robust testing.

- In the case of large datasets (e.g., thousands or millions of records), a test size of 10-20% is usually sufficient. With a large dataset, using more for testing can give a robust estimate of the model's performance.
- **train\_size**: we can specify how much of the data we want in the training set instead of specifying the test size.
  - Example: **train\_size=0.8** would mean that **80%** of the data is used for **training**.
- **random\_state**: This ensures the split is reproducible. Every time we run the code with the same **random\_state**, we will get the same train-test split. If we don't specify it, the split will be random each time you run the code.
  - **random\_state=42** is commonly used in tutorials and helps with consistency in results.
- **shuffle**: The data is shuffled before splitting by **default (shuffle=True)**. This ensures that the training and test sets are random and that any ordering present in the dataset (e.g., date, customer ID) doesn't affect the split.

## Example Code + Output:

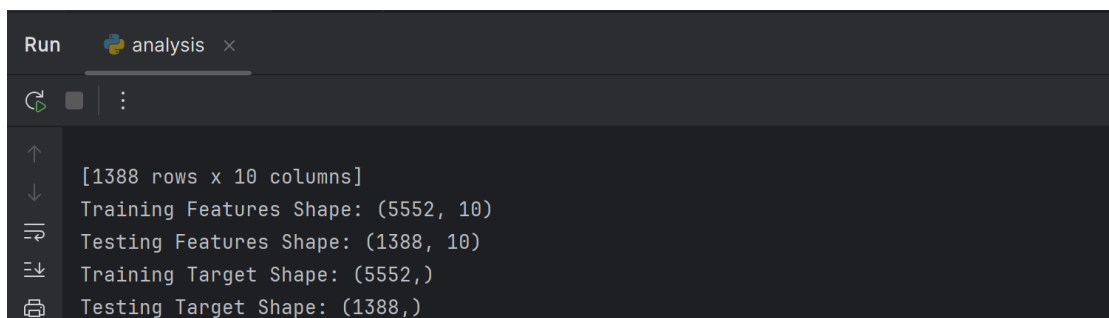
In below example, we want to split the dataset in a way that allows us to predict "Churn\_Yes" column. To do this, we exclude the target column ("Churn\_Yes") from the features (X) because it shouldn't be part of the inputs when building the model.

The features are used as input variables during model training. They are used to predict the target (output). If the target is included in the features, the model simply memorizes the values it supposed to predict, which is meaningless.

**Note:** The target variable is the dependent column that we want our machine learning model to predict based on the features (also called "independent variables").

```
# -----Step 5: Split the Data into Training and Test Sets
from sklearn.model_selection import train_test_split
X = data_encoded_cleaned.drop(labels='Churn_Yes', axis=1) # Features (everything except 'Churn_Yes')
y = data_encoded_cleaned['Churn_Yes'] # Target (the column we are trying to predict)
X_train, X_test, y_train, y_test = train_test_split(*arrays: X, y, test_size=0.2, random_state=42)
print("Training Features Shape:", X_train.shape)
print("Testing Features Shape:", X_test.shape)
print("Training Target Shape:", y_train.shape)
print("Testing Target Shape:", y_test.shape)
```

## Result:



```
Run analysis x
[1388 rows x 10 columns]
Training Features Shape: (5552, 10)
Testing Features Shape: (1388, 10)
Training Target Shape: (5552,)
Testing Target Shape: (1388,)
```

The output shows that **80%** of the data is used for training, and **20%** is held out for testing. In this case, there are **5552** samples in the training set and **1388** samples in the testing set.

After successfully splitting the dataset, we can:

- Use the training data to fit our model: This is where our model learns patterns from the data.
- Use the test data to evaluate the model: Test data serves as unseen data to assess how well our model generalizes.

During the training process, we use **X\_train** as input to train model and **y\_train** is used to help the model learn what “correct” means. After training, when we use the model to make predictions, it uses **X\_test** to generate predicted target values. These predicted target values are then compared to **y\_test** (the actual target values) to evaluate the model’s performance.

## Another Example Code + Output:

Depends on the nature of our problem or analysis, we can target any column:

- Target “**Churn\_Yes**” if we want to predict whether a customer will churn
- Target “**MonthlyCharges**” if we want to predict the monthly charges a customer will incur
- Target “**tenure**” if we want to predict the length of time a customer will stay with the company

We can target any column in our dataset as long as it is the one we want to predict. For example, now we want to predict “**MonthlyCharges**” instead of “**Churn\_Yes**”:

```
X = data_encoded_cleaned.drop(labels='MonthlyCharges', axis=1) # Features (everything except 'Churn_Yes')
y = data_encoded_cleaned['MonthlyCharges'] # Target (the column we are trying to predict)
X_train, X_test, y_train, y_test = train_test_split(*arrays: X, y, test_size=0.2, random_state=42)
print("Training Features Shape:", X_train.shape)
print("Testing Features Shape:", X_test.shape)
print("Training Target Shape:", y_train.shape)
print("Testing Target Shape:", y_test.shape)
```

### Result:

```
"D:\python\Customer churn rate analysis\.venv\Scripts\python.exe" "D:\python\Customer churn rate analysis\analysis.py"
Training Features Shape: (5552, 10)
Testing Features Shape: (1388, 10)
Training Target Shape: (5552,)
Testing Target Shape: (1388,)
```